Model-based Testing with Graph Transformation Systems

Vincent de Bruijn

December 18, 2011

Abstract

Model-based testing is one of the most used techniques to increase the quality of software. *On the fly* testing on symbolic models has proven itself to be a practical approach in testing large software systems. Another interesting modelling technique are graphs and rules for *Graph Transformations*. Graph-like models have many advantages, such as the familiarity people have with it. This report describes the setup of a research aimed at exploring the possibilities of using *Graph Transformation Systems* (GTS) in model-based testing. The goal is to create a system for automatic test generation on GTSs. An exisiting graph transformation tool and a model-based testing tool are used for this purpose. The design will be validated through several case-studies. The potential benefits of this research will be a new practical testing approach which includes the advantages of graph-based modelling and GTSs.

Contents

1 Introduction

Limited time and resources in software projects often make testing a neglected part of the software development process. The testing process is often done manually without proper automation, which quickly becomes very costly. Lack of proper software testing increases future costs, as errors become harder to repair when the system is put online.

The automation of the test process is a necessity, as testing should be done repeatedly throughout the development process. An exhaustive check of the system is often impossible, due to hardware restrictions. A widely used practice is maintaining a *test suite*, which contains tests for each system component. However, when the creation of a test suite is done manually, this still leaves room for human error and it becomes hard to judge how much of the system is tested, otherwise called the *completeness* of the test suite. Also, there is a trade-off between the completeness of the test suite and the time spent on creating tests manually.

Creating an abstract representation or a *model* of the system is a way to tackle these problems. A model can be used to systematically generate tests for the system. This is referred to as *model-based testing*. This leads to a larger test suite in a shorter amount of time than if done manually. These models are created from the specifications and requirements from the software developer, end-user and/or owner. The process of collecting the specification by the software developer into a model is still error-prone, due to imprecise, incomplete and ambiguous specifications. The visual or textual representation of large models becomes troublesome to understand for the end-user, which obstructs the feedback process.

A formalism that among other things can describe software systems are Graph Transformation Systems (GTS). These systems express the system state and transitions by means of graphs. This formalism may provide a more intuitive approach to system modelling. This is the motivation for the research described in this document: creating and validating a system that allows model-based testing on GTSs.

The structure of this document is as follows: first the research goals, the motivation for this research and the general approach are given in section 2. The model-based testing process is outlined in section 3. Sections 4 and 5 describe the relevant models. The tools used are introduced in section 6. The methods used in the research are in section 7. An overview of the related literature is made in section 8. Finally, conclusions are drawn in section 9. The planning for the research phase is in appendix A.

2 Research questions

In this section the research goals, a motivation for the research and the general approach to answering the research questions are given.

2.1 Questions

The research questions are split into a design and validation component:

1. Design: How can automatic test generation be done on graph transformation systems?
2. Validation: What are the strengths and weaknesses of using graph transformation systems in model-based testing?

The criteria used to assess the strengths and weaknesses are split into two parts: the objective and the subjective criteria. The objective criteria are the measurements that can be done on the implementation, such as speed and statespace size. The subjective criteria are related to how

easy graph-transformation models are to use and maintain. The assessment of the latter criteria requires a human component. The criteria and methods for the assessment are elaborated in section 7.2.

2.2 Motivation

A system for automatic test generation on graph transformation models is useful for software developers who require a testing tool that is easier to use with models that are easier to maintain. If the assumptions that GTSs provide a more intuitive modelling and testing process hold, this new testing approach will be less error-prone and will lead to less ambiguous specifications. The system design, once implemented and validated, provides a valuable contribution to the testing paradigm.

2.3 Approach

In order to show whether automatic test generation is possible on GTSs and how, a system will be created for automatic test generation on GTSs. The graph transformation tool GROOVE[1] will be used to model and explore the GTS. An already existing and used testing tool is developed by Axini[2]. This tool will be used to create and run the tests. It accepts a symbolic model, which is described in detail in section 4. Part of this research will be the design and implementation of the transformation of a GTS in GROOVE to a symbolic model in the Axini tool. The resulting system will be validated using several case-studies. These case-studies are done with existing specifications from systems tested by Axini. Each case-study will have a GTS and a symbolic model which describe the same system. The tool from the design part of the research and the Axini tool are used for the automatic test generation on these models respectively. Both the model and the test process will then be compared as part of the validation. This includes the criteria mentioned in section 2.1. This is explained in more detail in section 7.2.

3 Model-based Testing

Model-based testing is a testing technique where a System Under Test (SUT) is tested for conformance with a model that describes the system. The general setup for this process is depicted in Figure 1. The specification, which is given in the form of a model, is given to a test derivation component which generates the test cases. These test cases are passed to a component that executes the test cases on the SUT. This is done by providing input/stimuli to the SUT and monitoring the output/response. Based on the test cases, the stimuli and the response the test execution component shows whether the SUT conforms to the specification by giving a 'pass' or a 'fail' verdict.

3.1 Input-Output Transition Systems

A practical model for describing system behavior is the Input-Output Transition System (IOTS). An example of such a transition system is shown in Figure 2. This system allows an input of 20 or 50 cents and then outputs tea or coffee accordingly. The inputs are preceded by a question mark, the outputs are preceded by an exclamation mark.

[1] http://sourceforge.net/projects/groove/
[2] http://www.axini.nl/

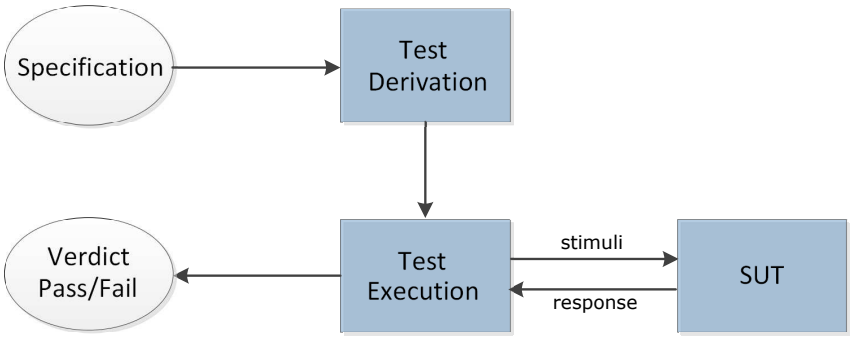


Figure 1: A general model-based testing setup

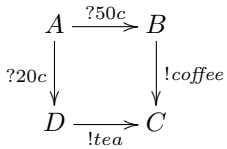


Figure 2: Example of an IOTS

A test case can also be described by an IOTS. A test case for the coffee machine is given in Figure 3. 50 cents is put into the machine, when the coffee output is observed from the system, the test passes, when the tea output is observed, the test fails.

4 Symbolic Transition Systems

Variations of state machines and transition systems have been widely used as the underlying model for test generation. Other tools use the structure of data types to generate test data. In this section, Symbolic Transition Systems (STS) are introduced which combine state transition systems with a data type oriented approach. These models do not suffer as quickly from the state-space explosion problem as Labelled Transition Systems do. They are used in practice in for instance the Axini tool.

4.1 Definition

The definition of an STS that follows is based on the definition given by Frantzen et al. [7]. The formalisms are then explained intuitively.
A Symbolic Transition System is a tuple $< L, l_0, \mathcal{V}, \imath, \mathcal{I}, \Lambda, \rightarrow >$, where:

- L is a countable set of locations and l_0 is the initial location.

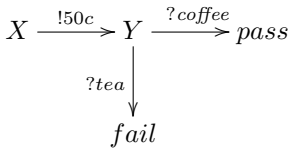


Figure 3: Example of an IOTS test case

- $\mathcal{V}$ is a countable set of location variables.
- $\imath$ is a set of mappings $\{v \mapsto x \mid v \in \mathcal{V}, x \in \mathcal{X}\}$ and $\mathcal{X}$ is a collection of ground terms, representing the initialisation of the location variables.
- $\mathcal{I}$ is a set of interaction variables, disjoint from $\mathcal{V}$.
- Λ is a finite set of gates. The unobservable gate is denoted $\tau(\tau \notin \Lambda)$; we write Λ_τ for $\Lambda \cup \{\tau\}$. The arity of a gate $\lambda \in \Lambda_\tau$, denoted $arity(\lambda)$, is a natural number. The parameters of a gate $\lambda \in \Lambda_\tau$, denoted $param(\lambda)$, is a tuple of length $arity(\lambda)$ of distinct interaction variables. We fix arity$(\tau) = 0$, i.e. the unobservable gate has no interaction variables.
- $\rightarrow \subseteq L \times \Lambda_\tau \times \mathcal{F}(\mathcal{V} \cup \mathcal{I} \cup \mathcal{X}) \times \{v \mapsto x \mid v \in \mathcal{V}, x \in \mathcal{G}(\mathcal{V} \cup \mathcal{I} \cup \mathcal{X})\} \times L$, where $\mathcal{F}$ is the collection of boolean functions over a set of variables and ground terms and $\mathcal{G}$ is the collection of mathematical functions over a set of variables and ground terms, is the switch relation. We write $l \xrightarrow{\lambda,\phi,\rho} l'$ instead of $(l, \lambda, \phi, \rho, l') \in \rightarrow$, where ϕ is referred to as the switch restriction (acting as a guard) and ρ as the update mapping. We require $var(\phi) \cup var(\rho) \subseteq \mathcal{V} \cup param(\lambda)$, where var is the collection of the variables used in the given guard or update mapping.

The locations can be seen as the states in the transition system. However, a state is referred to as a precise system state including the values of variables. This is not the case with locations, as they only express the place in the control flow where the system currently resides. Therefore there is a clear distinction between the system state and the system location. The location variables are *global* variables in the model, meaning the model keeps track of the values of these variables no matter in which location the system is in. They are specific to the model, meaning they do not keep track of memory values in the SUT. They have an initial value, which is given by the mapping in $\imath$. Gates are often called the action or label of a switch relation. However, gates also have accompanying interaction variables, listed in the *param* tuple. The arity of a gate is therefore the number of parameters of the gate. The switch relation allows the transition from one location to another location. This should not be confused with transitions, as these are referred to as the transition from one *state* to another state.

4.2 Example

In Figure 4 a simple board game is shown, where two players consecutively throw a die and move along four squares. The defining tuple of the STS is:
$< \{throw, move\}, throw, \{T, P, D\}, \{T \mapsto 0, P \mapsto [0, 2], D \mapsto 0\}, \{d, p, l\}, \{throw?, move!\},$
$\{throw \xrightarrow{trhow?, 1<=d<=6, D \mapsto d} move, move \xrightarrow{move!, T=p \wedge l=(P[p]+D)\%4, P[p] \mapsto l, T \mapsto p\%2} throw\} >$
The variables T, P and D are the location variables symbolizing the player's turn, the positions of the players and the number of the die thrown respectively. The gate *move*! has $param =< p, l >$ symbolizing which player moves to which location. The gate *throw*? has $param =< d >$ symbolizing which number is thrown by the die. The switch relation with gate *throw*? has the restriction that the number of the die thrown is between one and six and the update sets the location variable D to the value of interaction variable d. The switch relations with gate *move*! have the restriction that it must be the turn of the player moving and that the new location of the player is the number of steps ahead as thrown by the die. The update mapping sets the location of the player to the correct value and passes the turn to the next player. In the figure the first row shows the gate, the second row shows the constraint and the last row shows the update of the variables.

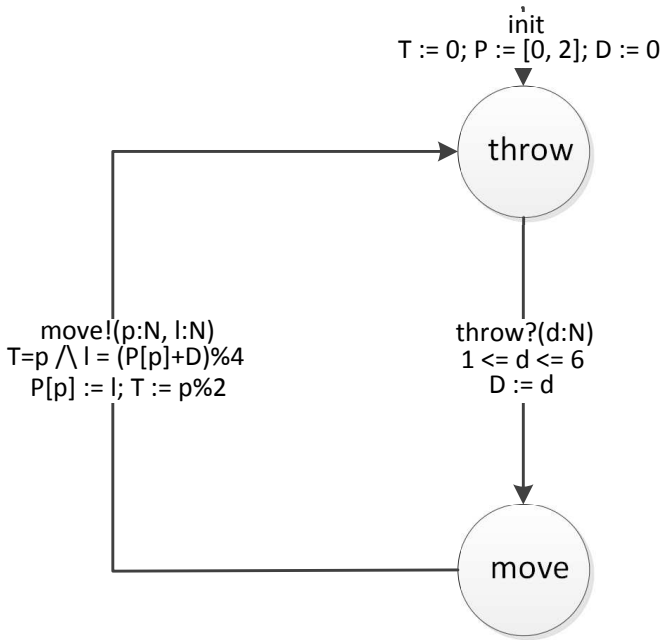


Figure 4: The STS of a board game example.

5 Graph Transformation Systems

A Graph Transformation System is composed of a start graph and a set of transition rules. The start graph describes the system in its initial state. The transition rules apply changes to the graph, creating a new graph which describes the system in its new state.

5.1 Definition

A Graph Transformation System is a tuple $< G, R >$, where:

- G is a graph of the start state $< L, N, E >$, where:
  - L is a set of labels
  - N is a set of nodes, where each $n \in N$ has a label $l \in L$
  - E is a set of edges, where each $e \in E$ has a label $l \in L$ and nodes $source, target \in N$
- R is a set of transformation rules (LHS, NAC, RHS), where:
  - LHS is a graph representing the left-hand side of the rule
  - NAC is a set of graphs representing the negative application conditions
  - RHS is a graph representing the right-hand side of the rule

A graph H is a *subgraph* of G, denoted $H \subseteq G$, if the node and edge sets of H are subsets of the node and edge sets of G, where nodes are considered equal if their labels coincide and edges are considered the same if their labels and their source and target nodes coincide. A rule R is applicable on a graph G if $LHS \subseteq G \wedge \neg \exists c \in NAC.c \subseteq G$. The resulting graph after the rule application is determined by RHS of R. The nodes and edges in LHS are mapped to equal nodes and edges in RHS. All elements in LHS not part of the mapping are removed from G, all elements in RHS not part of the mapping are added to G and all elements part of the mapping are kept.

5.2 Example

The running example is displayed as a GROOVE GTS model in Figure 5. Figure 5a is the start graph of the system. The rules can be described as follows:

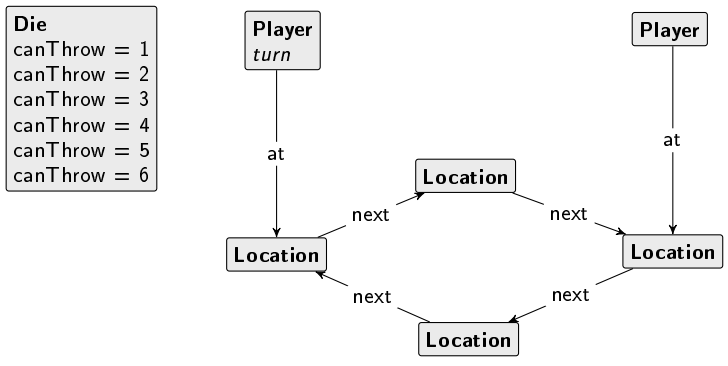
1. 5b: 'if a player has the turn and he has not thrown the die yet, he may do so.'
2. 5c: 'if a player has the turn and he has thrown the die and this number is larger than zero, he may move one place and then it is as if he has thrown one less.'
3. 5d: 'if a player has finished moving (number thrown is zero), the next player receives the turn.'

The *LHS*, *NAC* and *RHS* of each rule are displayed as one graph. The colored nodes and edges in the rules indicate to which part they belong:

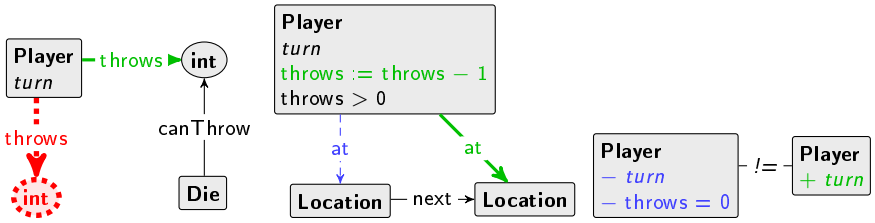
1. normal line (black): This node or edge is part of both the *LHS* and *RHS*.
2. dotted line (red): This node or edge is part of the *NAC* only.
3. thicker line (green): This node or edge is part of the *RHS* only.
4. dashed line (blue): This node or edge is part of the *LHS* only.

The *turn* flag on the *Player* node is a representation of a self-edge with label *turn*. The assignments on the *Die* node are representations of edges to integer nodes. The throws value assignment (:=) in the move rule represents an edge with the *throws* label to an integer node in the *LHS* and the same edge to another integer node in the *RHS*. In the next turn rule, the *turn* edge exists in the LHS as a self-edge of the left *Player* node and in the *RHS* as a self-edge of the right *Player* node. The *throws* edge from the left *Player* node to an integer node only exists in the *LHS*.

After the rule is applied, the graph is transformed according to the rule specifications. This is the new state of the system and the rule is the transition from the old state (the graph as it was before the rule was applied) to the new state.



(a) The start graph



(b) The throw rule (c) The move rule (d) The next turn rule

Figure 5: The GTS of the board game example in Figure 4

6 Tooling

6.1 Description Axini tool

The Axini tool is a web-based application, developed in the Ruby on Rails framework. The architecture is shown graphically in Figure 6. An STS is given to a Model Interpreter component, which passes information on the current state and possible transitions to the Test Engine component, which devises the test cases. The stimuli given to the SUT are based on the labels of the switch relations of the STS. In fact, this component also includes the strategy component that decides on the specific data values to test. The Test Engine gives the instantiated switch relation, which will be referred to as transition, to the Test Execution component as an *abstract stimulus*. The term abstract is used here to indicate that the label of the transition is an abstract representation of the actual stimulus given to the SUT. For instance, the label 'connect' is an abstract stimulus of the actual setup of a TCP connection between two distributed components of the SUT. This 'translation' is done by the Test Execution component. When the SUT responds, the Test Execution component translates this response back to an abstract response, in a similar fashion as described above. The Test Engine updates the Model Interpreter on which transition(s) were chosen and gives the pass or fail verdict based on the test case.

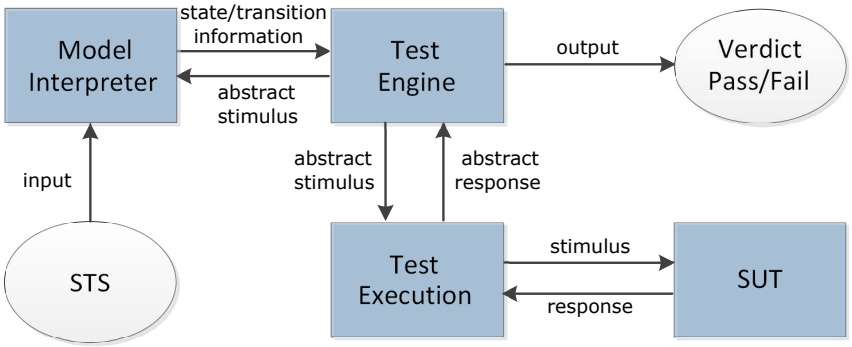


Figure 6: The Axini Tool

6.2 Description GROOVE

The architecture of the GROOVE tool is shown graphically in Figure 7. A Graph Transformation System is given as input to a Rule Applier component, which determines the possible rule matches. An Exploration Strategy can be started or the user can explore the states manually using the GUI. These components request the rule matches and give the chosen match as feedback. The Exploration Strategy can do an exhaustive search, leading to a Graph Transition System, which closely resembles an LTS, except the graph information is maintained in the state.

6.3 Comparison of the examples

The GTS of the running example in Figure 5 has a number of consecutive transitions when a player moves. This is different from the STS, which updates the location variable in one transition. The GTS can also model the location as a variable and update this variable in one transition. This model is shown in Figure 8. The movement of the players as modelled in the GTS can also be modelled in an STS. Figure 9 shows this model.

The new GTS loses many advantages by structuring it in this way: the overview of the board is gone, the rules are less visual and extending the locations in different directions is much harder.

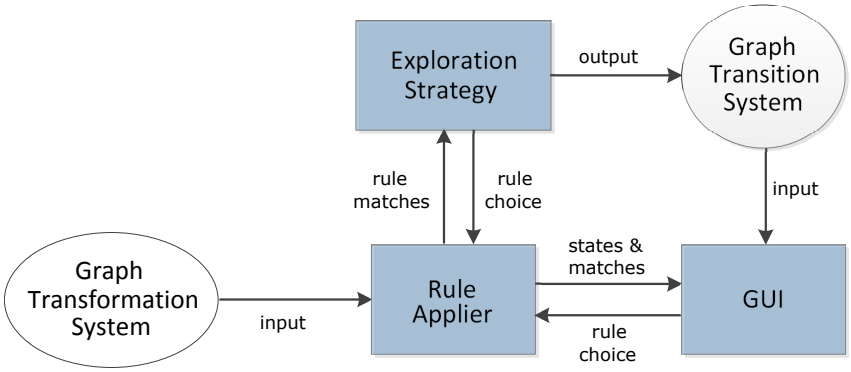
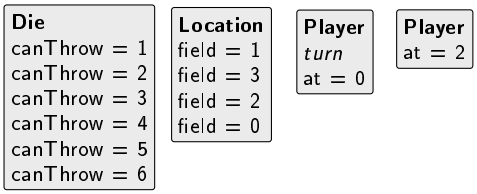
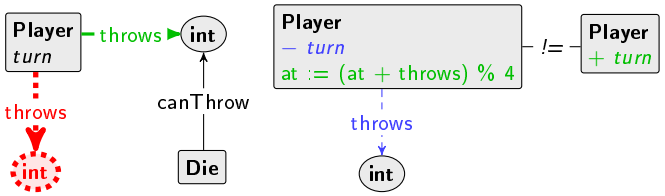


Figure 7: The GROOVE Tool



(a) The start graph



(b) The throw rule (c) The move rule

Figure 8: Another GTS model of the board game example

The rules do look more compact, there is even one rule less. However, when generating the Graph Transition Systems of both models with GROOVE a significant difference appears: the GTS from Figure 5 generates 32 states with 52 transitions, which can be seen visually in Figure 10, whereas the model from Figure 8 generates 112 states with 192 transitions (Figure 11). The reason is that the board is circular: to the first model, the players being at locations 1 and 3 is the same as them being at locations 2 and 4. However, this is not the same to the second model. This distinction results in a *symmetry reduction* in the first model and thus a smaller statespace than of the second model. In both models, another symmetry reduction takes place when the Players are at the same location.

The STS is also different than the one in Figure 4: now two switch relations are needed to move a player. The underlying LTS of the first model has 224 states and 384 transitions. This is calculated by taking all possibilities of the data values except for the die roll. This leads to 32 states ($4 \times 4 \times 2$). These 32 'throw' states each have 6 throw transitions to a 'move' state, thus there are 192 'move' states. The 'move' states only have one transition back to a 'throw' state. There are $6 \times 32 + 192 \times 1 = 384$ transitions. However, there is a possible bisimulation reduction. The specific value of D is not relevant anymore for the 'move' transitions in the LTS, as the constraint is already taken into account. Therefore the 'move' states that lead to the same 'throw' state can be grouped together. For each 'throw' state there is only one 'move' state, so this gives $32 \times 2 = 64$ states and $6 \times 32 + 32 \times 1 = 224$ transitions.

This reduction is not possible on the Graph Transition Systems, because state information is

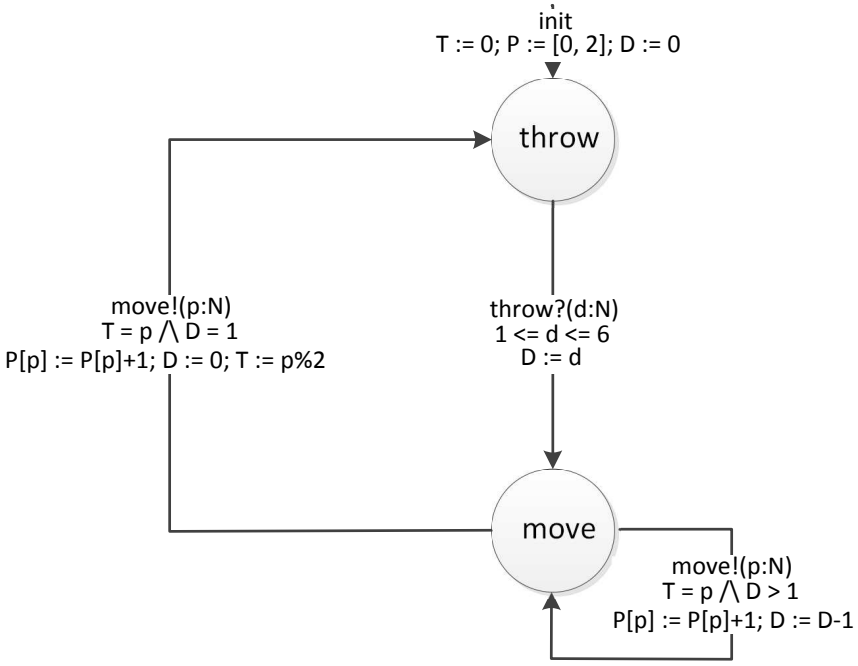


Figure 9: Another STS model of the board game example

maintained in the states. The underlying LTS (derived from the Transition System by ommitting the state information) does have this possible reduction. The symmetry reductions are possible on the STS, if more complex data values such as objects are allowed.

7 Research methods

7.1 Design

The specific method used to achieve the design goal is using GROOVE as a replacement of the symbolic model in the Axini tool. Figure 12 shows this graphically. The Rule Applier component now communicates with the Model Interpreter component. The communication runs through interfaces on both tools and through a socket over TCP, such that other tools like TorX are also able to communicate with the GROOVE component.

7.1.1 Problems

Already some problems with the implementation have been identified. They are listed in the following paragraph.

Modelling data types such as integers and strings in GROOVE is problematic; the range of possible values has to be given explicitly and cannot be infinite. For example, it would be impossible to extend the die of the running example such that it can throw any integer.
Repeated application of a rule in a GTS is sometimes a sign of one transition spread out across multiple transitions. In the case of the running example, this occurs with the player moving step by step.
The Axini tool annotates the locations it has been and the switch relations it has taken in the STS.

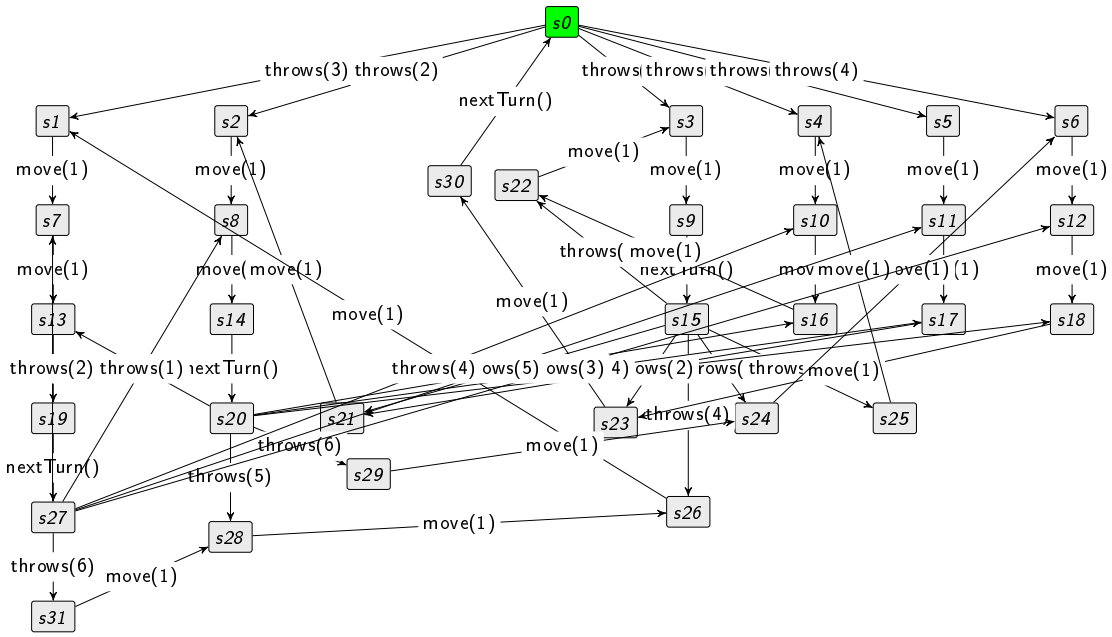


Figure 10: The Graph Transition System of the model in Figure 5

The GTS does not have these locations and switch relations (it does have states and transitions), so it is hard to annotate these on the GROOVE side. This annotation is used for coverage statistics.

7.1.2 Coverage

State, transition, location and switch relation coverage are already implemented in the Axini tool. Some form of location/switch relation coverage would also need to be implemented on the GROOVE tool. A representation of a location in GROOVE can be made by removing data values from a state graph. The applicable rules are a representation of the possible switch relations. Implementing this kind of 'graph/rule coverage' statistics is a possible field of research. An additional possible field of research is the implementation of data coverage statistics on the Axini Tool. This coverage statistic measures for each switch relation the representativeness of the data values used for all possible data values.

7.1.3 Design steps

The design steps will be done in increasing difficulty. In the next paragraph these steps are set out.

First, GROOVE will generate the entire LTS of the graph model and send it to the Axini tool for the test generation. The test engine in the Axini tool that accepts STSs is used, as this is the engine we want to modify. The LTS is also an STS, without variables, data values and contraints. The functionality of finding the underlying LTS of a GTS is already implemented in GROOVE. This will require the interface on both ends and the socket in between to be operational. The issue of limited data types should be solved at the end of this step.
Next, the interface on the GROOVE side will translate the GTS directly to an STS and transmit that instead of the LTS. This will require GTS-to-STS transformation rules. The issue that can be solved here is the grouping of transitions.
The final step is to implement a GTS engine on the Axini side and communicate on-the-fly as the

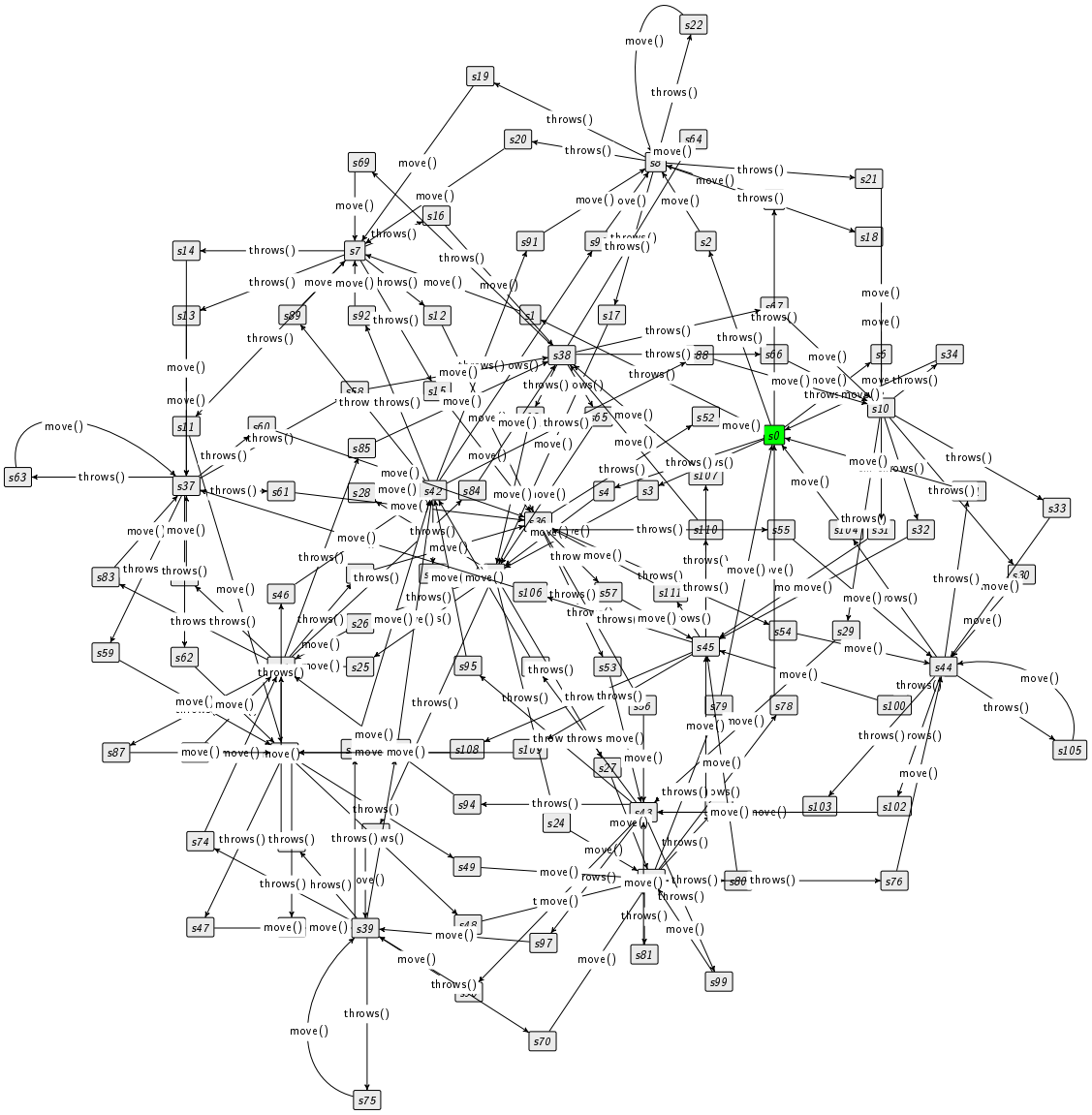


Figure 11: The Graph Transition System of the model in Figure 8

statespace of the GTS is explored on the GROOVE side. The final issue of location annotation for coverage statistics should be solved at the end of this step.

7.2 Validation

The first step in validating the newly created tool is by testing a small model, namely the running example. A SUT is made for this board game and the Axini tool is used to test this game. Then, the GROOVE-Axini tool is used to test the game. The results should reveal no errors, fail verdicts or other differences in the output of both tools. An intentional error is then made in the SUT and the process is repeated. Still no errors or differences are expected, but both tools should find the error and give a fail verdict. Next, the assessment of the strengths and weaknesses of the resulting GROOVE-Axini tool is done by applying both tools to several case studies and comparing the

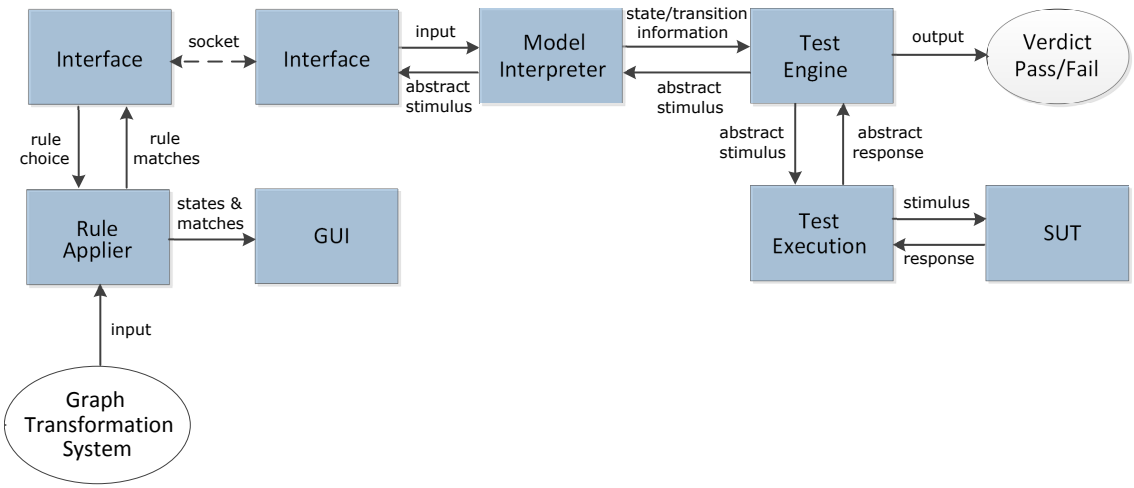


Figure 12: Replacing the symbolic model with GROOVE

results. The case studies are set out first and then the criteria for the comparison are given.

7.2.1 Case studies

One modelling technique might prove more practical for one kind of system, while the other modelling technique for another kind of system. Therefore, three case studies are planned. They are all real-life systems Axini has worked on:

- a self-scan register
- a Tom Tom navigation system
- a Philips health-care system

The self-scan register is a machine that automates the purchase of products at a supermarket. A customer can put his products on a conveyer belt and the system automatically calculates the price of the products. Then the customer pays and gets a receipt. The Tom Tom navigation system is a GPS device with a route planner. It allows a user to enter a destination and the system plans the correct route accordingly from the locatoion of the user. The Philips health-care system is some sort of medical device.

7.2.2 Objective criteria

As mentioned in section 2.1, the criteria are split in two parts. The objective criteria that can be compared are:

1. The verdicts
2. The bugs found
3. The coverage
4. The test cases generated per second
5. The size of the statespace

Same test cases should give same verdicts. When a verdict is different for the same test case in the GROOVE-Axini tool and the Axini tool, this might indicate an error. Another possibility is

that one tool generates smarter test-cases and finds more bugs or reaches a higher coverage. The tools are also benchmarked on how fast they generate test-cases and on how much space they use.

7.2.3 Subjective criteria

This research will not include an extensive experiment with a statistical significant set of human actors. However, a group of experts in the field will be invited to a discussion at the end of the research period. This will include students and doctorates from the Formal Methods department of the University of Twente and employees from Axini. Preceding the discussion, the case studies and comparisions are presented. The discussion is meant for the participants to become acquainted with both modelling techniques and share ideas and thoughts. After the discussion, the participants are asked to work on one of the case studies. This work entails either:

1. Extending the model with a new feature
2. Finding a bug/error in the SUT/model

The following results are then noted and compared:

1. The time spent on the assignment
2. The correctness of the result
3. The feelings the participants had with the assignment (how difficult, how much fun, etc.)

With these results also the affiliation of the participants is taken into account: people from Axini and GROOVE are expected to do a better in their respective fields. Therefore, they will also be able to compare both modelling processes when working with the other tool than the one they are already familiar with.

The results will give insight in the understandibility, maintainability and extendibility of both modelling processes. The results are compiled and presented in the final thesis.

8 Related work

8.1 Model-based testing

Formal testing theory was introduced by De Nicola et al.[8]. This testing theory was first used in algorithms for automatic test generation by Brinksma[2]. The specification language LOTOS was used in this research as a defining notation for transition systems. Tretmans gives a formal approach to protocol conformance testing (whether a protocol conforms to its specifications) in [11]. A useful transition system for model-based testing is the input-output transition system by Tretmans[12], assuming that implementations communicate with their environment via inputs and outputs. Additional work on the conformance testing framework for LTS-based testing with inputs and outputs has been done by van der Bijl and Peureux[13].
A tool for the automatic synthesis of test cases for nondeterministic systems is TGV[6]. Another tool, TorX, integrates automatic test generation, test execution, and test analysis in an on-the-fly manner[10].

8.2 Symbolic

Symbolic test generation is introduced by Rusu et al. [9], using Input-Output Symbolic Transition Systems (IOSTSs). Symbolic Transition Systems (STSs) are introduced by Frantzen et al. [7]. A

tool that generates tests based on symbolic specification is the STG tool, introduced in Clarke et al. [3].

8.3 Graph transformations

Graph transformations have been studied since the early '70s. The usage of this computational paradigm is best described by the following quote from Andries et al. [1]: "Graphs are well-known, well-understood, and frequently used means to represent system states, complex objects, diagrams, and networks, like flowcharts, entity-relationship diagrams, Petri nets, and many more. Rules have proved to be extremely useful for describing computations by local transformations: Arithmetic, syntactic, and deduction rules are well-known examples." An informative paper on graph transformations is written by Heckel et al.[5]. A quote from this paper: "Graphs and diagrams provide a simple and powerful approach to a variety of problems that are typical to computer science in general, and software engineering in particular." These quotations supports the assumptions made in section 2.
The modelling tool GROOVE has been applied to several case studies, such as model transformations and security and leader election protocols [4].

9 Conclusion

This report motivates the need of a research towards a model-based testing practice on Graph Transformation Systems. The research will investigate whether the assumption that GTSs will provide a more understandable, easier practice is true. The goal will be to create a tool that allows automatic test generation on GTSs and assess the strengths and weaknesses of this test practice.

The first results in this report demonstrate that GTSs can provide a nice overview of a system. The results also show an interesting automatic statespace reduction, namely the symmetry reduction. The second GTS of the example shows that also a purely arithmic model can be built; this indicates the strength of the formalism.

The design phase is split into small steps that should break down the complexity of the entire implementation process. The validation with the use of the case studies emphasises the practicality of the tooling; the purpose of the test tools is to be used on real-world software systems. Finally, the experiments are a great indication of the usefulness of the tool towards software testers.

It is my hope that the tool produced will be used in practice and that it will provide a better modelling and testing experience for its users.

References

[1] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-JÃűrg Kreowski, Sabine Kuske, Detlef Plump, Andy SchÃijrr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1 – 54, 1999.

[2] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45510-8_9.

[3] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Stg: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the*

Construction and Analysis of Systems, volume 2280 of *Lecture Notes in Computer Science*, pages 151–173. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-46002-0_34.

[4] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using groove. *International journal on software tools for technology transfer*, online pre-publication, March 2011.

[5] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1 SPEC. ISS.):187–198, 2006. cited By (since 1996) 16.

[6] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 7:297–315, 2005. 10.1007/s10009-004-0153-x.

[7] Tim A.C. Willemse Lars Frantzen, Jan Tretmans. Test generation based of symbolic specifcations. Technical report, Nijmegen Institute for Computing and Information Sciences (NIII), 2005.

[8] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83 – 133, 1984.

[9] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-40911-4_20.

[10] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[11] Jan Tretmans. A formal approach to conformance testing, 1992.

[12] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61042-1_42.

[13] Machiel van der Bijl and Fabien Peureux. I/o-automata based testing. In *Model-Based Testing of Reactive Systems*, pages 173–200, 2004.

A Planning

When	What	Deliverable
week 1-3 2 Jan - 22 Jan	Implementing basic interfaces for both tools. Creating SUT of boardgame example.	Working system for automatic test generation on GTSs by generating LTS.
week 4-6 23 Jan - 12 Feb	Setting up transformation rules for GTS-to-STS. Implementing transformation on GROOVE interface.	1) GTS-to-STS transformation rules. 2) System for automatic test generation on GTSs by generating STS.
week 7-10 13 Feb - 11 Mar	Implementing GTS engine on the Axini tool and graph/rule coverage on the GROOVE tool.	System for automatic on-the-fly test generation on GTSs with coverage statistics.
week 11-13 12 Mar - 1 Apr	Creating GTSs for each case study.	Graph-based models of several software systems.
week 14 2 Apr - 8 Apr	Running the implemented system on the created models.	Measurements and comparison of both tools on objective criteria.
week 15-16 9 Apr - 22 Apr	Optional: Researching possibilities of data coverage + implementation.	Support for data coverage statistics in both tools.
week 17-19 23 Apr - 13 May	Setting up experiments with participants.	Buggy models, case descriptions, appointments for experiments.
week 20 14 May - 20 May	Running experiments.	Measurements and comparison of both tools on subjective criteria.
week 21-23 21 May - 10 Jun	Finish writing thesis.	The final thesis.